Global Predicate Detection
and Event Ordering

Our Problem
To compute predicates
over the state of
a distributed application

Model
- Message passing
- No failures
- Two possible timing assumptions:
  1. Synchronous System
  2. Asynchronous System
     - No upper bound on message delivery time
     - No bound on relative process speeds

Asynchronous systems
- Weakest possible assumptions
cfr. “finite progress axiom”
- Weak assumptions ⊑ less vulnerabilities
- Asynchronous ≠ slow
- “Interesting” model w.r.t. failures (ah ah ah!)
Client-Server

Processes exchange messages using Remote Procedure Call (RPC)

A client requests a service by sending the server a message. The client blocks while waiting for a response.

Deadlock!

Goal

Design a protocol by which a processor can determine whether a global predicate (say, deadlock) holds
Draw arrow from $p_i$ to $p_j$ if $p_j$ has received a request but has not responded yet.

Cycle in WFG $\Rightarrow$ deadlock

Deadlock $\Rightarrow$ cycle in WFG

The protocol

$p_0$ sends a message to $p_1 \ldots p_3$

On receipt of $p_0$'s message, $p_i$ replies with its state and wait-for info

An execution
An execution

Houston, we have a problem...

- Asynchronous system
  - no centralized clock, etc. etc.
- Synchrony useful to
  - coordinate actions
  - order events
- Mmmmmhh...

Events and Histories

- Processes execute sequences of events
- Events can be of 3 types: local, send, and receive
- $e_p^i$ is the $i$-th event of process $p$
- The local history $h_p$ of process $p$ is the sequence of events executed by process $p$
- $h_p^k$: prefix that contains first $k$ events
- $h_p^0$: initial, empty sequence
- The history $H$ is the set $h_p^0 \cup h_p^1 \cup \ldots \cup h_p^{n-1}$

Note: In $H$, local histories are interpreted as sets, rather than sequences, of events.
Ordering events

Observation 1:
Events in a local history are totally ordered

Observation 2:
For every message \( m \), \( \text{send}(m) \) precedes \( \text{receive}(m) \)

Happened-before (Lamport[1978])

A binary relation \( \preceq \) defined over events

1. If \( e_i^k, e_i^l \in h_i \) and \( k < l \), then \( e_i^k \rightarrow e_i^l \)
2. If \( e_i = \text{send}(m) \) and \( e_j = \text{receive}(m) \), then \( e_i \rightarrow e_j \)
3. If \( e \rightarrow e' \) and \( e' \rightarrow e'' \) then \( e \rightarrow e'' \)

Space-Time diagrams

A graphic representation of a distributed execution
Space-Time diagrams

A graphic representation of a distributed execution

H and \( \rightarrow \) impose a partial order
Space-Time diagrams

A graphic representation of a distributed execution

\[ p_1 \rightarrow p_2 \rightarrow p_3 \]

\[ p_1 \rightarrow p_2 \rightarrow p_3 \]

H and \( \rightarrow \) impose a partial order

Runs and Consistent Runs

A run is a total ordering of the events in H that is consistent with the local histories of the processors

\[ \text{Ex: } h_1, h_2, \ldots, h_n \text{ is a run} \]

A run is consistent if the total order imposed in the run is an extension of the partial order induced by \( \rightarrow \)

A single distributed computation may correspond to several consistent runs!
**Cuts**

A cut $C$ is a subset of the global history of $H$

$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$

**Global states and cuts**

- The **global state** of a distributed computation is an $n$-tuple of local states
  
  $\Sigma = (\sigma_1, \ldots, \sigma_n)$

- To each cut $(c_1 \ldots c_n)$ corresponds a global state $(\sigma_1^{c_1}, \ldots, \sigma_n^{c_n})$

**Consistent cuts and consistent global states**

- A cut is consistent if
  
  $\forall e_i, e_j : e_j \in C \land e_i \rightarrow e_j \Rightarrow e_i \in C$

- A **consistent global state** is one corresponding to a consistent cut
Our task

- Develop a protocol by which a processor can build a consistent global state
- Informally, we want to be able to take a snapshot of the computation
- Not obvious in an asynchronous system...

Our approach

- Develop a simple synchronous protocol
- Refine protocol as we relax assumptions
- Record:
  - processor states
  - channel states
- Assumptions:
  - FIFO channels
  - Each message timestamped with $T(send(m))$
Snapshot I

i. \( p_0 \) selects \( t_{ss} \)
ii. \( p_0 \) sends "take a snapshot at \( t_{ss} \)" to all processes
iii. when clock of \( p_i \) reads \( t_{ss} \) then
   a. records its local state \( \sigma_i \)
   b. starts recording messages received on each of incoming channels
   c. stops recording a channel when it receives first message with timestamp greater than or equal to \( t_{ss} \)

Correctness

Theorem
Snapshot I produces a consistent cut

Proof
Need to prove \( e_j \in C \land e_i \rightarrow e_j \Rightarrow e_i \in C \)

< Definition >
0. \( e_j \in C \equiv T(e_j) < t_{ss} \)
3. \( T(e_j) < t_{ss} \)
6. \( T(e_i) < t_{ss} \)

< Assumption >
1. \( e_j \in C \)
4. \( e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j) \)
7. \( e_i \in C \)

< Assumption >
2. \( e_i \rightarrow e_j \)
5. \( T(e_i) < T(e_j) \)

< Property of real time>
4. \( e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j) \)

Clock Condition

Can the Clock Condition be implemented some other way?
Lamport Clocks

Each process maintains a local variable \( LC \)

\[ LC(e) \equiv \text{value of } LC \text{ for event } e \]

\[ LC(e_i^p) < LC(e_{i+1}^p) \]

\[ LC(e_i^p) < LC(e_q^j) \]

Increment Rules

\[ LC(e_i^p) = LC(e_{i-1}^p) + 1 \]

\[ LC(e_q^j) = \max(LC(e_{q-1}^j), LC(e_i^p)) + 1 \]

Timestamp \( m \) with \( TS(m) = LC(send(m)) \)

Space-Time Diagrams and Logical Clocks

A subtle problem

when \( LC = t \) do \( S \) doesn't make sense for Lamport clocks!

\( S \) is anyway executed after \( LC = t \)

Fixes:

- if \( e \) is internal/send and \( LC = t - 2 \)
  - execute \( e \) and then \( S \)
- if \( e = receive(m) \land (TS(m) \geq t) \land (LC \leq t - 1) \)
  - put message back in channel
  - re-enable \( e \); set \( LC = t - 1 \); execute \( S \)
An obvious problem

No \( t_{ss} \)!

Choose \( \Omega \) large enough that it cannot be reached by applying the update rules of logical clocks

\[ \text{mmmmhhhh...} \]

Doing so assumes

- upper bound on message delivery time
- upper bound relative process speeds

We better relax it...

An obvious problem

No \( t_{ss} \)!

Choose \( \Omega \) large enough that it cannot be reached by applying the update rules of logical clocks

An obvious problem

No \( t_{ss} \)!

Choose \( \Omega \) large enough that it cannot be reached by applying the update rules of logical clocks

\[ \text{mmmmhhhh...} \]

Snapshot II

processor \( p_0 \) selects \( \Omega \)

\( p_0 \) sends “take a snapshot at \( \Omega \)” to all processes; it waits for all of them to reply and then sets its logical clock to \( \Omega \)

When clock of \( p_i \) read \( \Omega \) then \( p_i \)

- records its local state \( \sigma_i \)
- sends an empty message along its outgoing channels
- starts recording messages received on each incoming channel
- stops recording a channel when receives first message with timestamp greater than or equal to \( \Omega \)
Relaxing synchrony

Process does nothing for the protocol during this time!

Use empty message to announce snapshot!

Snapshots: a perspective

The global state $\Sigma^*$ saved by the snapshot protocol is a consistent global state

But did it ever occur during the computation?

- a distributed computation provides only a partial order of events
- many total orders (runs) are compatible with that partial order
- all we know is that $\Sigma^*$ could have occurred
Snapshots: a perspective

- The global state $\Sigma^g$ saved by the snapshot protocol is a consistent global state.
- But did it ever occur during the computation?
  - a distributed computation provides only a partial order of events.
  - many total orders (runs) are compatible with that partial order.
  - all we know is that $\Sigma^g$ could have occurred.
- We are evaluating predicates on states that may have never occurred!

An Execution and its Lattice

An Execution and its Lattice
An Execution and its Lattice
An Execution and its Lattice

\[ \Sigma_{00} \Sigma_{01} \Sigma_{02} \]

\[ \Sigma_{10} \Sigma_{11} \Sigma_{12} \]

\[ \Sigma_{20} \Sigma_{21} \Sigma_{22} \]

\[ \Sigma_{30} \Sigma_{31} \Sigma_{32} \]
An Execution and its Lattice

Reachability

$\Sigma^k$ is reachable from $\Sigma^i$ if there is a path from $\Sigma^k$ to $\Sigma^i$ in the lattice
Reachability

\( \Sigma^k \) is reachable from \( \Sigma^l \) if there is a path from \( \Sigma^k \) to \( \Sigma^l \) in the lattice.

So, why do we care about \( \Sigma^s \) again?

- Deadlock is a stable property.
  - Deadlock \( \Rightarrow \) Deadlock
- If a run \( R \) of the snapshot protocol starts in \( \Sigma^i \) and terminates in \( \Sigma^f \), then \( \Sigma^i \sim_R \Sigma^f \).
So, why do we care about $\Sigma^s$ again?

- Deadlock is a stable property
  - Deadlock $\Rightarrow \square$ Deadlock
- If a run $R$ of the snapshot protocol starts in $\Sigma^i$ and terminates in $\Sigma^f$, then $\Sigma^i \sim_R \Sigma^f$
- Deadlock in $\Sigma^s$ implies deadlock in $\Sigma^f$
- No deadlock in $\Sigma^s$ implies no deadlock in $\Sigma^i$